UNIVERSITATEA TEHNICĂ
DIN CLUJ-NAPOCA

TEHNICI DE PROGRAMARE

ORDER MANAGEMENT
TEMA 3

DOCUMENTAȚIE

Student: Nemes Emilia
Grupa: 30228
Profesor: Pop Cristina

Cuprins

1. Obiectivul temei ……3
1.1 Obiectivul principal……...3
1.2 Obiectivele secundare……....3
2. Analiza problemei……4
2.1 Cazuri de utilizare……4
3. Proiectare……....5
3.1 Diagramă UML……...5
3.1.1 Diagramă de pachete.…….5
3.1.2 Diagrame de clase.…….5
3.2 Proiectare pachete…….7
3.3 Structuri de date……...8
3.4 Modul de tratare a erorilor…….8
4. Implementare……...8
4.1 Pachetul *main*…….8
4.2 Pachetul *presentation* ……...……....8
4.3 Pachetul *connection*……...……....9
4.4 Pachetul *model*……...……....9
4.5 Pachetul *dataAccessLayer*……...……....9
4.6 Pachetul *businessLayer*……...……....10
4.7 Pachetul *bll.validators*……...……....11
5. Rezultate……...……...11
6. Concluzii……...……..11
7. Bibliografie……...……..11

1. Obiectivul temei

1.1 Obiectivul principal

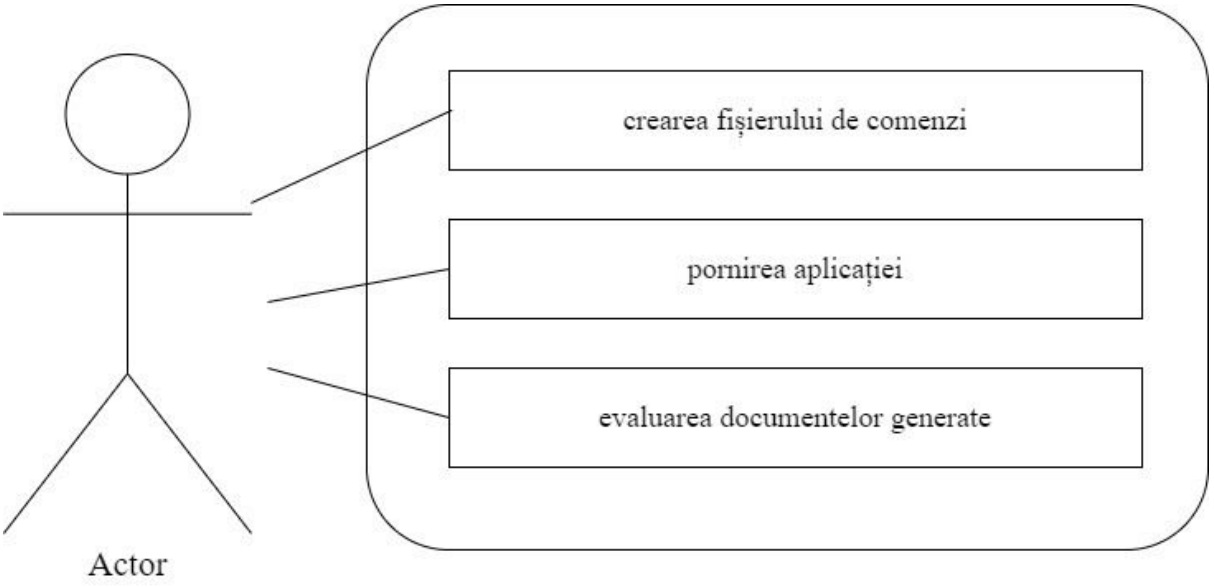
Proiectaţi o aplicaţie numită Order Management care procesează comenzile clienţilor pentru un depozit. Pentru stocarea produselor, clienţilor şi a comenzilor se folosesc baze de date relaţionale.
Aplicaţia ar trebui să permită procesarea comenzilor dintr-un fişier text dat ca argument, efectuarea operaţiilor specificate, salvarea informaţiilor în baza de date şi generarea rapoartelor într-un format pdf.

1.2 Obiectivele secundare

Nr.	Obiectivul secundar	Descriere	Capitolul
1.	Dezvoltarea unui caz de utilizare	Cazul de utilizare descrie secvenţa de interacţiune între utilizator şi aplicaţie	2
2.	Decizii de proiectare	Decizii asupra modului de proiectare cu scopul de implementa proiectul	3
3.	Proiectarea pachetelor	Enumerarea pachetelor implementate şi descrierea rolurilor acestora	3
4.	Alegerea structurilor de date	Structuri de date folosite în cadrul proiectului	3
5.	Modul de tratare a erorilor	Cum am tratat erorile care pot să apară	3
6.	Implementarea soluţiei	Realizarea proiectului prin cod, detalii despre acesta	4

2. Analiza problemei

2.1 Cazuri de utilizare



Aplicaţia realizează managementul comenzilor, care permite stocarea datelor într-o bază de date relaţională, şi totodată permite şi prelucrarea datelor acestora (precum inserarea, căutarea, updatarea, ştergerea).
Actorul, adică utilizatorul acestei aplicaţii are posibilitatea de a efectua diferite operaţii (care sunt enumerate într-un fişier text) cu ajutorul bazei de date. Se pot introduce sau şterge clienţi şi produse, se pot updata produsele, se pot face comenzi de către clienţi înregistraţi şi se pot face rapoarte pentru a vizualiza toţi clienţii, produse sau ordere prezente în baza de date la momentul respectiv.

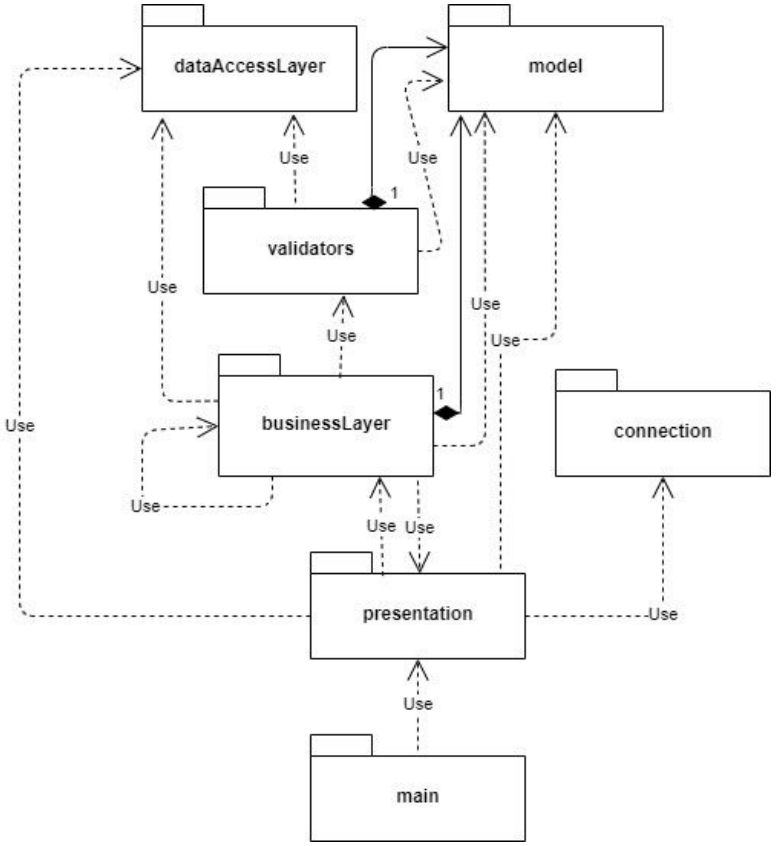
Descrierea cazurilor de utilizare:

- se face un fişier numit *commands.txt* care conţine operaţiile care trebuie efectuate
- se face un fişier numit *dump.txt* care conţine codul SQL pentru crearea tabelelor necesare pentru stocarea datelor din fişierul *commands.txt*
- utilizatorul introduce de pe linia de comandă numele fişierului de input (în cazul nostru *commands.txt*) care conţine operaţiile de efectuat (insert, delete, order, report)
- se începe programul care procesează operaţiile specificate, şi prelucrează datele prin intermediul bazei de date
- se generează fişiere de output în cazul în care se fac comenzi (factura pentru comandă sau mesaj de eroare cu client neînregistrat sau under-stock pentru produs) sau în cazul operaţiilor de raport (conţine toate item-urile din tabela corespunzătoare pentru client, produs sau order)

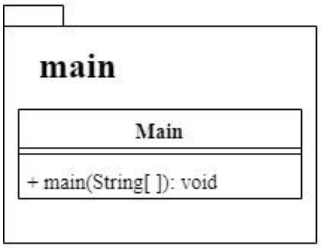
3. Proiectare

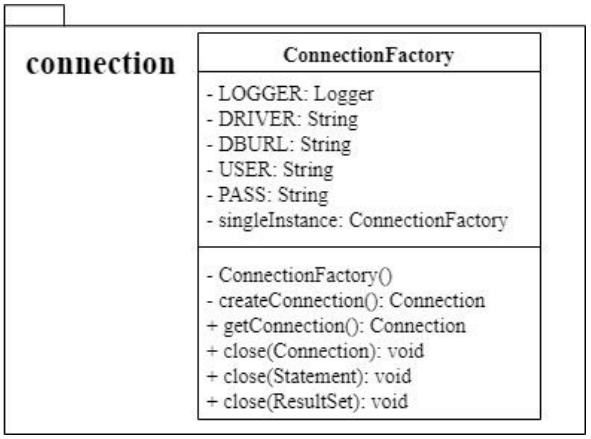
3.1 Diagramă UML

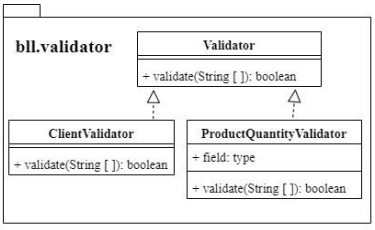
3.1.1 Diagramă de pachete

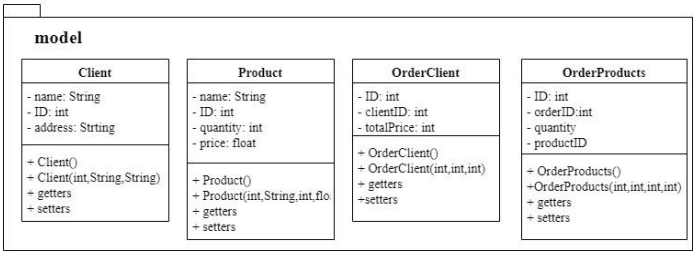


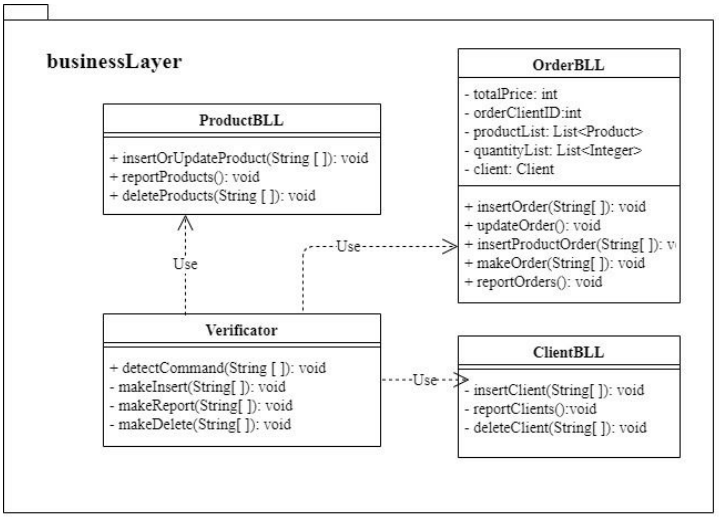
3.1.2 Diagrame de clase

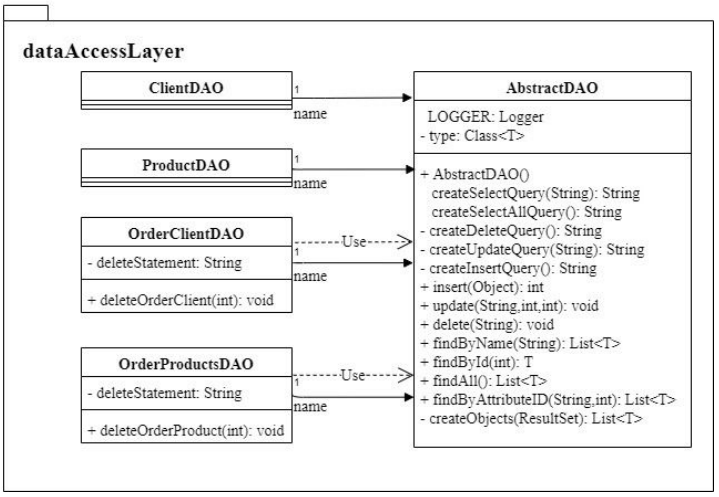


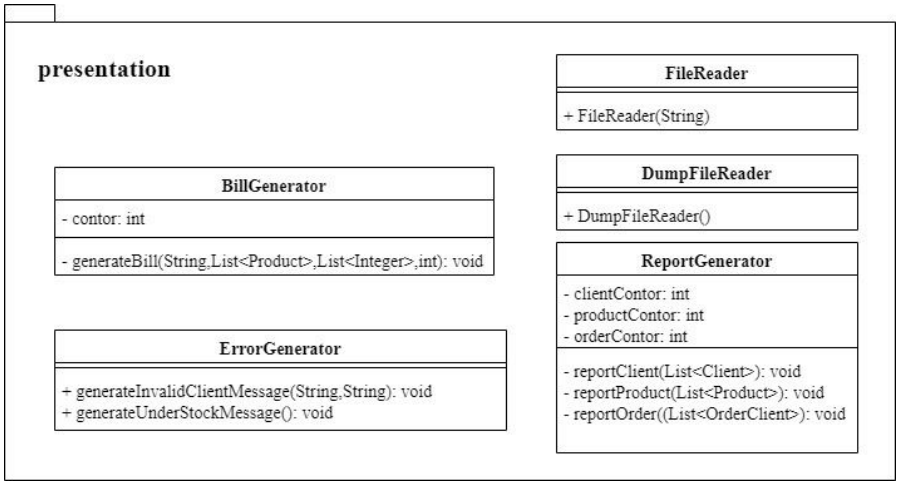



bll.validator
Validator
+ validate(String [ ]): boolean
ClientValidator
+ validate(String [ ]): boolean
ProductQuantityValidator
+ field: type
+ validate(String [ ]): boolean


model
Client
- name: String
- ID: int
- address: Strting
+ Client()
+ Client(int,String,String)
+ getters
+ setters
Product
- name: String
- ID: int
- quantity: int
- price: float
+ Product()
+ Product(int,String,int,flo
+ getters
+ setters
OrderClient
- ID: int
- clientID: int
- totalPrice: int
+ OrderClient()
+ OrderClient(int,int,int)
+ getters
+setters
OrderProducts
- ID: int
- orderID:int
- quantity
- productID
+ OrderProducts()
+OrderProducts(int,int,int,int)
+ getters
+ setters


businessLayer
ProductBLL
+ insertOrUpdateProduct(String [ ]): void
+ reportProducts(): void
+ deleteProducts(String [ ]): void
OrderBLL
- totalPrice: int
- orderClientID:int
- productList: List<Product>
- quantityList: List<Integer>
- client: Client
+ insertOrder(String[ ]): void
+ updateOrder(): void
+ insertProductOrder(String[ ]): v
+ makeOrder(String[ ]): void
+ reportOrders(): void
Use
Use
Verificator
+ detectCommand(String [ ]): void
- makeInsert(String[ ]): void
- makeReport(String[ ]): void
- makeDelete(String[ ]): void
Use
ClientBLL
- insertClient(String[ ]): void
- reportClients():void
- deleteClient(String[ ]): void


dataAccessLayer
ClientDAO
1
name
AbstractDAO
LOGGER: Logger
- type: Class<T>
+ AbstractDAO()
createSelectQuery(String): String
createSelectAllQuery(): String
- createDeleteQuery(): String
- createUpdateQuery(String): String
- createInsertQuery(): String
+ insert(Object): int
+ update(String,int,int): void
+ delete(String): void
+ findByName(String): List<T>
+ findById(int): T
+ findAll(): List<T>
+ findByAttributeID(String,int): List<T>
- createObjects(ResultSet): List<T>
ProductDAO
1
name
OrderClientDAO
- deleteStatement: String
+ deleteOrderClient(int): void
Use
1
1
name
OrderProductsDAO
- deleteStatement: String
+ deleteOrderProduct(int): void
Use
1
name



3.2 Proiectare pachete

Diagrama pachetelor şi a claselor, împreună cu dependenţele dintre ele, este prezentată în figura de la secţiunea 3.1 Diagramă UML. Aceasta arată legăturile dintre pachetele, care sunt de tip de compoziţie şi de folosire (use).

După cum se vede din diagrame în realizarea proiectului am implementat şapte pachete, fiecare având un scop bine definit. Pachetul *main* conţine clasa *Main*, care are o singură metodă, numită *main*, din care se începe rularea aplicaţiei. Pachetul *presentation* conţine clasele care realizează citirea din fişiere (precum clasele *FileReader* şi *DumpFileReader*) şi clasele care realizează scrierea în fişiere (precum clasele *ReportGenerator*, *BillGenerator* şi *ErrorGenerator*). Pachetul *connection* conţine clasa *ConnectionFactory* care realizează conectarea programului la baza de date. Pachetul *model* conţine clasele care modelează aplicaţia, având clasele *Client*, *Product*, *OrderClient* şi *OrderProducts*. Pachetul *dataAccessLayer* realizează comunicarea cu baza de date, prin definirea şi implementarea statement-elor SQL. Pachetul conţine cinci clase: *AbstractDAO*, care conţine toate statement-urile şi implementările lor în mod general, şi clasele *ClientDAO*, *ProductDAO*, *OrderClientDAO* şi *OrderProductsDAO* care extind clasa *AbstractDAO* implementând metodele ei pe obiecte de tipul corespunzător lor. Astfel se foloseşte tehnica de reflection în accesarea bazei de date. Pachetul *businessLayer* implementează logica aplicaţiei, şi conţine clase precum: *Verificator*, *ClientBLL*, *ProductBLL*, *OrderBLL*. Clasa *Verificator* parsează liniile din fişierul de input cu operaţii, verificând ce trebuie efectuată: inserare, update, comandă, ştergere sau generarea unui raport. Celelalte clase din acest pachet implementează metodele corespunzătoare acestor operaţii. Dar pentru a genera output-uri corecte ele apelează metode din pachetul *bll.validators*. Pachetul acesta implementează o interfaţă *Validator*, care are o metodă *validate()*. Celelalte două clase din pachet, anume clasele *ClientValidator* şi *ProductQuantityValidator* implementează interfaţa *Validator* şi metoda ei *validate()*, pentru a verifica dacă există în baza de date client cu numele specificat în order respectiv există cantitatea necesară din produse pentru a realiza comanda.

3.3 Structuri de date

În cazul atributelor folosesc diferite tipuri de date: tipul primitiv *int*, pentru atribute care stochează valori întregi (precum ID, cantitate); tipul primitiv *float*, pentru atributul care stochează valoare reală (în cazul preţului); tipul *String,* pentru atribute care stochează şiruri de caractere(precum nume, adresă). În cazul implementării statement-elor SQL am folosit tipul de date *StringBuilder* pentru a le construi mai uşor din părţi separate (de ex. fielduri după care se face căutarea, semnele de întrebare etc.).
Pe lângă acestea mai folosesc structuri de date necesare pentru lucrul cu fişiere, precum *File*, *Scanner*, *Document*, *Font, Paragraph* şi *Chunk*. Dar şi structuri de date necesare pentru lucrul cu baza de date, precum: *Logger*, *Connection*, *Statement*, *ResultSet*.

3.4 Modul de tratare a erorilor

Erorile pot să apară în cazul în care comanda nu poate fi realizată. Ori din cauză că nu există în baza de date clientul care face comanda, ori în cazul în care nu există pe stoc cantitatea de produs comandată. În cazul în care motivul erorii este clientul invalid se generează un document pdf cu numele *OrderError_Client* în care se specifică faptul că nu există clientul cu numele specificat în baza de date. Iar în cazul în care eroare este cauzată de cantitatea mică de produs se generează un document pdf cu numele *OrderError_Product* în care se afişează un mesaj de under-stock.

4. Implementare

4.1. Pachetul *main*

Pachetul *main* conţine o singură clasă numită *Main*. Clasa aceasta implementează metoda *main(String[]),* din care se instanţiază un obiect de tipul clasei *DumpFileReader* şi unul de tipul clasei *FileReader*. Obiectul de tip *DumpFileReader* citeşte dintr-un fişier instrucţiunile SQL pentru a crea tabelele necesare în baza de date, iar de tip *FileRead* citeşte operaţiile(commands) care trebuie efectuate.

4.2. Pachetul *presentation*

Pachetul *presentation* conţine clasele *DumpFileReader*, *FileReader* şi *ReportGenerator*, *BillGenerator, ErrorGenerator*, care sunt fişiere de intrare, respectiv fişiere de ieşire.
Clasa *DumpFileReader* are un constructor, în care se citesc dintr-un fişier statement-urile pentru a crea tabelele bazei de date. Crearea tabelelor prin cod SQL uşurează modificarea bazei de date în caz în care se mai introduc alte aspecte în comenzi, şi este nevoie de alte tabele la nivelul bazei de date. Citirea codului se face linie cu linie din fişierul de dump, după care se execută unul după altul fiecare instrucţiune, prin intermediul conexiunii cu baza de date.
Clasa *FileReader* are în mod asemănător cu clasa *DumpFileReader* un singur constructor, în care se citesc instrucţiunile din fişierul cu commands linie cu linie. Fiecare linie este o instrucţiune separată, de aceea se evaluează fiecare linie pe rând de către metoda *detectCommand* a clasei *Verificator* pentru a detecta ce operaţii trebuie efectuate. Liniile citite din fişierul de input sunt parsate la rândul lor pe cuvinte, astfel fiecare linie se stochează într-un vector de String-uri numit *strings*.
Clasa *ReportGenerator* are trei metode, câte una pentru generarea raportului de client, produs şi order.
Metodele creez documente noi de tip pdf, în care se fac tabele, care au coloane corespunzătoare atributelor claselor (mai puţin în cazul claselor *OrderClient* şi *OrderProducts* care compun împreună tabelul din raportul order-ului). Tabelele sunt populate cu datele din tabelele bazei de date

corespunzătoare claselor *Client*, *Product*, *OrderClient* şi *OrderProducts*. Metoda are totodată şi trei atribute de tip *static* care sunt contori pentru documentele generate, ca să aibă fiecare un nume diferit.
Clasa *BillGenerator* are un atribut numit *contor* de tip int, care este unică pe clasă (deci este *static*), şi care este concatenat la finalul numelui documentului generat, numit *Bill*. Prin introducerea valorii contorului în numele fişierului se face o distincţie între documentele generate, şi se evită suprascrierea lor, în cazul în care se generează mai multe facturi în cadrul aceluiaşi fişier de instrucţiuni (*commands.txt*). După ce se generează un document cu numele specificat anterior, se deschide documentul şi se afişează în el numele clientului care a făcut comanda, produsele comandate de el, şi cantitatea comandată din fiecare, şi preţul total al comenzii, care este o sumă între preţul tuturor produselor comandate.
Clasa *ErrorGenerator* implementează două metode *generateInvalidClientMessage(String, String)* şi *generateUnderStockMessage()*. Prima generează un document pdf cu numele *OrderError_Client* în care se specifică faptul că nu există client în baza de date cu numele specificat în comandă. A doua generează un document pdf cu numele *OrderError_Product* în care se afişează un mesaj de under-stock.

4.3. Pachetul *connection*

Pachetul *connection* conţine o singură clasă numită *ConnectionFactory*. Clasa aceasta are scopul de a crea legătura între program şi baza de date. Are metode precum *createConnection(), getConnection(), close(Statement), close(ResultSet)*. De fiecare dată când programul modifică ceva în baza de date trebuie să facă o conexiune cu aceasta, iar după ce a terminat operaţiile trebuie să închidă conexiunea, prin metodele enumerate.

4.4. Pachetul *model*

Pachetul *model* conţine clasele *Client*, *Product*, *OrderClient* şi *OrderProducts*. Clasele acestea au fiecare un tabel corespondent în baza de date, cu care poate comunica.
Clasa *Client* are trei atribute: *ID* de tip *int*, *name* şi *address* de tip *String*, fiecare client fiind caracterizat de un nume unic şi de o adresă. Atributul *ID* este util numai când se citesc datele unui client din baza de date, pentru a putea stoca ID-ul din din baza de date al clientului respectiv.
Clasa *Product* are patru atribute: *ID* şi *quantity* de tip *int*, *name* de tip *String*, şi *price* de tip *float*, fiecare produs fiind caracterizat de un nume unic, o cantitate (ce se găseşte pe stoc) şi de un preţ unitar. ID-ul are rolul din baza de date, ca şi în cazul clienţilor.
Clasa *OrderClient* are trei atribute: *ID, clientID* şi *totalPrice* toate de tip *int*. Obiectele de acest tip (precum şi tabela corespunzătoare din baza de date) stochează numai ID-ul clientului care a făcut comanda, şi valoarea preţului total al comenzii.
Clasa *OrderProducts* are însă patru atribute: *ID, orderID, productID* şi *quantity* toate de tip *int*, şi face legătură cu tabela *OrderClient* prin câmpul orderID. Mai multe item-uri din tabela *OrderProducts* paote să conţină acelaşi orderID, astfel un order poate să conţină mai multe produse. Câmpurile *productID* şi *quantity* decsriu produsul din comandă şi cantitatea comandată din el.

4.5. Pachetul *dataAccessLayer*

Pachetul *dataAccessLayer* conţine clasele *AbstractDAO*, *ClientDAO*, *ProductDAO*, *OrderClientDAO* şi *OrderProductsDAO.*
Clasa *AbstractDAO* conţine implementările fiecărui statement SQL, necesare pentru prelucrarea datelor din baza de date. Acestea sunt însă implementate în mod abstract, generalizat. Ea conţine metode de două tipuri: metodele *createSelectQuery(String), createSelectAllQuery(), createDeleteQuery(), createUpdateQuery(String)* şi *createInsertQuery(),* care construiesc statement-urile pentru operaţiile de select, delete, update şi insert. Cealaltă categorie sunt metodele: *insert(Object), update(String,int,int), delete(String), findByName(String), findById(int), findAll(), findByAttributeID(String,int)* şi

createObjects(ResultSet), care execută statement-urile definite şi construiesc obiecte din datele returnate de către baza de date.

Tehnica de reflection se realizează prin faptul că celelalte clase din acest pachet extind clasa *AbstractDAO*, şi implementează metodele ei cu obiecte de tipul precizat de fiecare.

Clasa *ClientDAO* extinde clasa *AbstractDAO* cu parametru de tip *Client,* şi nu implementează nicio altă metodă adiţională.

Clasa *ProductDAO* extinde clasa *AbstractDAO* cu parametru de tip *Product,* şi nu implementează nicio altă metodă adiţională.

Clasa *OrderClientDAO* extinde clasa *AbstractDAO* cu parametru de tip *OrderClient,* şi implementează o metodă numită *deleteOrderClient(int)* care şterge din baza de date item-urile din tabela *OrderClient* cu ID-ul specificat ca parametru. Clasa *AbstractDAO* are metodă de ştergere doar pe baza numelui, dar nu şi pe baza ID-ului, de aceea era necesară implementarea acestei metode.

Clasa *OrderProductsDAO* extinde clasa *AbstractDAO* cu parametru de tip *OrderProducts,* şi implementează o metodă numită *deleteOrderProduct(int)* care şterge din baza de date item-urile din tabela OrderProducts care au orderID-ul specificat ca parametru.

4.6. Pachetul *businessLayer*

Pachetul *businessLayer* conţine clasele *Verificator*, *ClientBLL*, *ProductBLL* şi *OrderBLL*.

Clasa *Verificator* are patru metode: *detectCommand(String[]), makeInsert(String[]), makeReport(String[])* şi *makeDelete(String[])*. Metoda *detectCommand(String[])* primeşte ca parametru vectorul de String cu instrucţiunea ce trebuie efectuată, citită din fişierul *commands.txt*. Ea evaluează ce fel de operaţie este cerută (insert, order, report sau delete), şi baza acestuia apelează una din celelalte metode, sau metoda *makeOrder*(*String[])* din clasa *OrderBLL*. Celelalte trei metode verifică la care element se referă operaţia (client, produs sau order), şi apelează metodele corespunzătoare din celelalte clase ale pachetului acesta.

Clasa *ClientBLL* conţine trei metode: *insertClient(String[]), reportClients()* şi *deleteClient(String[])*. Prima metodă instanţiază un obiect nou de tipul Client cu valorile enumerate în instrucţiune îl inserează în baza de date prin apelearea metodei *insert(Client)* din clasa *ClientDAO*. A doua metodă caută toţi clienţii din baza de date, apelând metoda *findAll()* din clasa *ClientDAO,* şi generează un raport cu acestea, prin apelarea metodei *reportClient(List<Client>)* din clasa *ClientDAO.* A treia metodă şterge clientul specificat în intrucţiune din baza de date, dar mai întâi caută comenzile efectuate de acel client, şi şterge toate comenzile lui (se şterg item-urile din tabelele *OrderClient* şi *OrderProducts*), şi doar după acesta se şterge şi clientul din tabela *Client.*

Clasa *ProductBLL* conţine metodele: *insertOrUpdateProduct(String[]), reportProducts()* şi *deleteProduct(String[])*. La fel ca şi în clasa *ClientBLL* se execută operaţiile necesare pentru a face insert, delete sau report. Singura diferenţă este că pe produse se pot face şi update-uri: dacă apare în lista de instrucţiuni inserarea unui produs care a fost deja inserat, prin inserarea lui se înţelege updatarea stocului lui. Deci cantitatea produsului se modifică: va avea valoare cantităţii vechi adunat cu cantitatea din instrucţiunea nouă de *insert*.

Clasa *OrderBLL* conţine clasele: *insertOrder(String[]), updateOrder(), insertProductOrder(String[]), makeOrder(String[]), reportOrders()*. Deoarece comenzile (order) sunt stocate fizic în două tabele a bazei de date este necesar prelucrarea datelor acelor tabele în cazul în care se face vreo operaţie cu ordere. Când se face un order nou se apelează metoda *makeOrder(String[]),* care verifică dacă clientul care face comanda este valid, şi dacă este pe stoc cantitatea cerută din fiecare produs. În cazul în care nu se întâmpină astfel de erori, se apelează mai întâi metoda *insertOrder(String[]),* pentru a stoca ID-ul clientului care a făcut comanda, după care se apelează metoda *insertProductOrder(String[]),* în care se pot stoca caracteristicile produselor comandate, şi şi ID-ul orderului din tabela *OrderClient*. După ce se introduc toate item-urile în *OrderProducts* (numărul item-urilor inserate la o comandă corespunde cu numărul produselor comandate) se updatează *totalPrice*-ul din tabela *OrderClients,* care nu a fost

calculat inițial când s-a făcut inserarea în acest tabel, fiindcă produsele comandate au fost parcurse numai după. Metoda *reportOrders()* generează un raport cu toate comenzile, prin apelarea metodei *reportOrder(List<OrderClient>)* din clasa *OrderClientDAO*.

4.7. Pachetul *bll.validators*

Pachetul *bll.validators* conține interfața *Validator* și clasele *ClientValidator* și *ProductQuantityValidator,* care implementează ambele interfața *Validator*. Interfața aceasta are o singură metodă numită *validate(String[])*. Clasa *ClientValidator* implementează metoda respectivă astfel încât caută dacă clientul primit ca parametru există sau nu în baza de date, iar clasa *ProductQuantityValidator* implementează metoda prin verificarea dacă cantitatea pe stoc a tuturor produselor comandate este sau nu mai mare sau egală cu cantitățile comandate din produse. Metoda returnează un *boolean,* care este *true* sau *false* pe baza evaluării condițiilor.

5. Rezultate

Fișierele de ieșire conțin datele corespunzătoare command-urilor, sau mesaje de eroare în cazul orderelor invalide. Datele cu care s-a lucrat pot fi verificați și în baza de date, starea lor corespunde cu operațiile efectuate.

6. Concluzii

Aplicația implementată este una care realizează managementul comenzilor, modelând funcționarea unui depozit de produse. Clienții pot fi înregistrați, șterși, se pot introduce produse noi sau să se updateze stocurile, și se pot face comenzi de clienții înregistrați, în limita stocurilor disponibile.
Aplicația poate fi dezvoltată prin introducerea a noi câmpuri în tabelele existente, prin introducerea a noi tabele pentru a stoca și altfel de informații, sau prin adăugarea altor funcționalități, precum trimiterea newsletter-urilor cu produse noi.

7. Bibliografie

http://tutorials.jenkov.com/java-reflection/index.html
https://www.baeldung.com/java-pdf-creation
https://www.baeldung.com/javadoc
https://dev.mysql.com/doc/refman/5.7/en/using-mysqldump.html